CS 118 Computer Networks Fundamentals

Project 2: Window-based Reliable Data Transfer over UDP

Fall 2014

Student Name: He Ma

UID: 904434330

SEASnet login: sunnymh0728

Student Name: Qianwen Zhang

UID: 004401414

SEASnet login: qwzhang

Overview

In this project, we implement a window-based reliable data transfer built on top of Go-Back-N protocol with and without congestion control over UDP. The client will first send a packet requesting for a file. If the file exists, the server will divide the entire file into small segments and send over the segments as packets with header.

Implementation

1. Protocol

To ensure reliable data transfer, the server uses Go-Back-N protocol with and without congestion control. Packets are allowed to sent in pipeline as long as no more than N packets are sent but not yet ACKed.

On the client side, the receiver adds data to the file and sends an ACK to the server if the packet received is in order and not corrupted; The receiver drops the packet and sends ACK if the packet is out of order or corrupted. Accumulative ACK is used. Because sequence number starts from 0 in our implementation, ACK k means that the first k bytes of the file is received and packet with sequence number k is expected.

On the server side, two parameters `base` and `next_seq_number` are used to mark the bytes. `[0, base-1]` are the bytes sent and ACKed. `[base, next_seq_number-1]` are the bytes sent but not yet ACKed. `[next_seq_number, file_size-1]` are the bytes to be sent. the sender slides the window down by 1 `DATA_SIZE` when a right ACK is received. We need to make sure that `next_seq_number - base <= window_size`. For GBN, `window_size` is fixed. For GBN with congestion control, `window_size` is updated as time goes by.

2. Packet format

We use struct with the fields below to denote the packets:

```
struct packet {
    int type;
    int seq_no;
    int length;
    int fin;
    char data[DATA_SIZE];
}
```

`type`: 0 - request, 1 - data, 2 - ACK, 3 - corrupted

`seq_no:` sequence number of current packet (data and ACK packets only)
`length:` length of data field carried in the packet (request and data packets only)
`data[DATA_SIZE]:` data that to be sent (request and data packets only)
`fin`: whether it is the last packet (data packet only)
`DATA_SIZE` is 1000 bytes by default. So the total size of each packet is 1016 bytes.

3. Message format
Whenever a packet is received or sent, its information is printed in the console.
Request packet:
`[Timestamp][SENT/RECV] Request [len(file_path)]: [file_path]`
Data packet:
`[Timestamp] [SENT/RECV] Data [sequence number] [len(data))`
ACK packet:
`[Timestamp] [SENT/RECV] ACK [sequence number]`
Corrupted packet:
`[Timestamp] [SENT/RECV] CORRUPT`
Time out and error are also printed to the console when they occur.
For congestion control, cwnd and ssthresh are printed to the console when they are updated.

4. Timeout event
We used `timer` and `select` functions to achieve the timeout checkings. We take down the time when the timer starts and the time before calling `recvfrom`. Then we can set the timeout for `recvfrom` as:
(`Max_timeout_time - current_time+time_start_time`).
The timeout for the client is 1 second. The timeout for the server is 0.1 second.

5. Extra credit
We implemented slow start, congestion avoidance and fast recovery. `cwnd` is set as 1 and `ssthresh` as 6 at the beginning. `cwnd` doubles its size each time during slow start. `cwnd` increases by 1 during congestion avoidance. If any of the ACKs is duplicated 3 times, `ssthresh=max(2,cwnd/2)` and `cwnd=ssthresh+3`. And everything in the window is resend.

Manual

1. Go to the directory containing server.c, client.c and packet.c
2. Compile with:

```
$ make
```
3. Run the server using:
```
$ ./server <port_number> <enable_congestion_control>
```
Run the client using：
```
$ ./client <sender_hostname> <sender_port_number>
  <filename> <enable_congestion_control>
```

Note:
* Congestion control is used only when congestion control is enabled for both sides.
* All files in server are stored in server_data directory and all files in client are stored in client_data directory.

Example:
To transfer `./server_data/subdirectory/test.txt`, run:
```
$ ./server 55555 1
$ ./client localhost 55555 1 subdirectory/test.txt
```
You will find the transferred file in `client_data` as `./client_data/test.txt`

Difficulties

1. Request packet not received or server shutdown
It is possible that the request packet is lost or corrupted. And it is also possible that the server is shutdown during the file transfer. We don't want the client to wait forever for the rest of the data packets. So we make the client to shut down if it doesn't receive any packet from socket for more than 1 second.

2. Client shutdown
It is possible that the client shuts down during the file transfer. Again, we don't want the server to wait forever for the next ACK message. To handle this case, we make the server stop sending data packets if it ever has to resend the same packet more than 100 times.

3. Last ACK from client
We should not close the client immediately after the client receive the last data packet because it is possible that the last ACK packet is lost. In this case the server will think that the client didn't receive the last data packet and keep resending it. To handle this case, we make the client shut down only if it doesn't receive any more packet in the

next 1 second. If it does receive a packet from the server, send the last ACK packet again.

4. File path size and file size

Because we are only sending 1 request packet for each file transfer request, the file path size shouldn't be bigger than the `DATA_SIZE` for packet. In our implementation, it is 999 bytes. Also we used `int` for tracking sequence number, so we can't handle file size bigger than 2^32 bytes, which is around 2GB. Additionally, we don't think it makes sense to transfer a file of size 0. It is easier to create an empty file in client directly.

5. '\0' for C char*

C char array should be terminated with '\0'. When we allocate the data field as 1000 in the packet, we can only put a string with length 999 in it. So the actual data size is 999. We we set the window size, it should be a multiple of 999 rather than 1000. Also, when we are writing to file, we should not include the '\0' in the file. So we are only writing 999 bytes to file each time. It caused us a lot of trouble when implemented all the I/Os.